Table of Contents

Introduction	1.1
介绍	1.2
文件名称	1.3
文件组织	1.4
缩进	1.5
注释	1.6
声明	1.7
语句	1.8
空白	1.9
命名规范	1.10
编程惯例	1.11
代码示例	1.12

Code Conventions for the Java Programming Language 中文翻译《Java 编码规范》

Chinese translation of the Code Conventions for the Java Programming Language document. This document contains the standard conventions that we at Sun follow and recommend that others follow. It covers filenames, file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices and includes a code example.

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

The last revision to this document was revised and updated on April 20, 1999 by Sun Microsystems. There is alse a GitBook version of the book: http://waylau.gitbooks.io/java-code-conventions/ or http://waylau.com/java-code-conventions/ Let's READ!

《Java 编码规范》中文翻译。这个文档包含了 Sun 所遵循以及推荐的标准的规范。它涵盖了文件名，文件组织，压缩，声明，注释，语句，空白，命名规范，编码惯例以及一个代码示例。

- 一个软件的生命周期中，80%的花费用于维护.
- 几乎没有任何一个软件，在其整个生命周期中，均由最初的作者来维护.
- 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码.

文档最后版本由 Sun 公司于1999年4月20日更新，虽然过去十几年，但这个规范至今对编码的规范指导仍然适用，特此翻译。如有勘误欢迎指正。

从目录开始阅读吧

Get Started 如何开始阅读

选择下面入口之一：

- https://github.com/waylau/java-code-conventions 的 SUMMARY.md（源码）
- http://waylau.gitbooks.io/java-code-conventions 点击 Read 按钮（同步更新，国内访问速度一般）
- http://waylau.com/java-code-conventions（国内访问速度快，定期更新。最后更新于2017-8-31）

Issue 意见、建议

如有勘误、意见或建议欢迎拍砖 https://github.com/waylau/java-code-conventions/issues

Contact 联系作者:

- Blog: waylau.com
- Gmail: waylau521(at)gmail.com
- Weibo: waylau521
- Twitter: waylau521
- Github : waylau

1 - 介绍

1.1 为啥要有编码规范

对于编程人员来说，编码规范的重要性体现在以下几个方面:

- 一个软件的生命周期中，80%的花费用于维护.
- 几乎没有任何一个软件，在其整个生命周期中，均由最初的作者来维护.
- 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码.
- 如果你把你的源代码作为一个产品，你要确保它是否被很好的打包并且清晰无误，就像你已构建的其它任何产品一样.

1.2 致谢

本文档反映的是 Sun MicroSystems 公司 Java Language Specification（Java语言规范） 中的编码标准部分,主要贡献者有 Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath, 和 Scott Hommel.

CONTENTS | NEXT

2 - 文件名称

这部分列出了常用的文件名及其后缀

2.1 文件后缀

Java 软件使用下面文件后缀:

文件类别	后缀
Java 源文件	.java
Java 字节码	.class

2.2 常见文件名称

常见文件名称包括:

文件名称	用法
`GNUmakefile`	makefiles 的首选文件名。我们使用 `gnumake` 来构建软件.
`README`	概述特定目录下所含内容的文件的首选文件名.

PREVIOUS | CONTENTS | NEXT

3 - 文件组织

一个文件由被空行分割而成的段落以及标识每个段落的可选注释共同组成。

超过2000行的程序难以阅读，应该尽量避免。

正确编码格式的范例，见 "Java 源文件案例".

3.1 Java 源文件

每个 Java 源文件都包含一个单一的公共类或接口。若私有类和接口与一个公共类相关联，可以将它们和公共类放入同一个源文件。公共类必须是这个文件中的第一个类或接口。

Java 源文件还遵循以下规则：

- 开头注释
- 包和引入语句
- 类和接口声明

3.1.1 开头注释

所有的源文件都应该在开头有一个C语言风格的注释，其中列出类名、版本信息、日期和版权声明:

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

3.1.2 包和引入语句

在多数 Java 源文件中，第一个非注释行是 `package` 语句。在它之后可以跟 `import` 语句。例如：

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

3.1.3 类和接口声明

下表描述了类和接口声明的各个部分以及它们出现的先后次序。见 "Java 源文件案例" 一个包含注释的例子.

class/interface 声明的各个部分	注解
class/interface 文档注释 (`/**...*/`)	详见 "文档注释"
`class` 或 `interface` 声明	
class/interface 实现注释 (`/*...*/`)，如果有必要的话	该注释应包含任何有关整个类或接口的信息，而这些信息又不适合作为 class/interface 文档注释。
class (`static`) 变量	先是 `public` class 变量，接着是 `protected` ，再是包级别 (没有访问修饰符)，再是 `private` 。
实例变量	先是 `public` class 变量，接着是 `protected` ，再是包级别 (没有访问修饰符)， 再是 `private` 。
构造器	
方法	这些方法应该按功能，而非作用域或访问权限分组。例如，一个私有的类方法可以置于两个公有的实例方法之间，其目的是为了更便于阅读和理解代码。

PREVIOUS | CONTENTS | NEXT

4 - 缩进

4 个空格常被作为缩进排版的一个单位。本文档并没有规定缩进的实现方式 (可以使用空格或者Tab建)。如果使用Tab，则Tab设置为 8 个空格 (而非 4 个)。

译者注：不同的IED或者文本编辑器中，Tab的空格数是不同的，常见的有4格或者8格。所以，这个文档中，如果强调了使用 Tab 键来缩进，意味着空了8格。否则，直接说明空了几个格，不然，会产生歧义。至于为什么是8格，我个人理解，这个规范应该是参考 C 语言规范而来。一般不建议用Tab键来代替空格，除非整个团队都有一致的开发工具或者编码格式文件。

4.1 行长度

尽量避免一行的长度超过 80 个字符，因为很多终端和工具不能很好处理。

注意**:** 用于文档中的例子应该使用更短的行长，长度一般不超过 70 个字符。

4.2 换行

当一个表达式无法容纳在一行内时，可以依据如下一般规则断开:

- 在一个逗号后面断开
- 在一个操作符前面断开
- 宁可选择较高级别 (higher-level) 的断开，而非较低级别 (lower-level) 的断开
- 新的一行应该与上一行同一级别表达式的开头处对齐
- 如果以上规则导致你的代码混乱或者使你的代码都堆挤在右边，那就代之以缩进8个空格。

以下是断开方法调用的一些例子:

```
someMethod(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);

var = someMethod1(longExpression1,
                someMethod2(longExpression2,
                        longExpression3));
```

以下是两个断开算术表达式的例子。前者更好，因为断开处位于括号表达式的外边，这是个较高级别的断开。

```
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longname6; // 推荐

longName1 = longName2 * (longName3 + longName4
                         - longName5) + 4 * longname6; // 避免
```

以下是两个缩进方法声明的例子。前者是常规情形。后者如果按照常规的缩进方法就会使得第二行和第三行太靠右边，所以只缩进8个字符。

```
//常规缩进
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

//缩进8个空格来避免缩进的太深
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}
```

if 语句的换行通常使用 8 个空格的规则，因为常规缩进(4个空格)会使语句体看起来比较费劲。比如:

```
//不要使用这种
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();            //MAKE THIS LINE EASY TO MISS
}

//用这个来代替
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//或用这种
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

这里有三种可行的方法用于处理三元运算表达式:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

PREVIOUS | CONTENTS | NEXT

5 - 注释

Java 有两类注释: implementation comments（实现注释）和 documentation comments（文档注释）。实现注释常见于 C++，使用 `/*...*/` 和 `//` 。文档注释 (也称为"doc comments") 是 Java 独有的，使用 `/**...*/` 。文档注释可以通过 javadoc 工具转成 HTML 文件。

实现注释用以注释代码或者特殊的实现。 文档注释从 implementation-free (实现自由)的角度描述代码的规范。它可以被那些手头没有源码的开发人员读懂。

注释应被用来给出代码的总览，并提供代码自身没有提供的附加信息。注释应该仅包含与阅读和理解程序有关的信息。例如，相应的包如何被建立或位于哪个目录下之类的信息不应包括在注释中。

在注释里，对设计决策中重要的或者不是显而易见的地方进行说明是可以的，但应避免提供代码中已清晰表达出来的重复信息。多余的的注释很容易过时。通常应避免那些代码更新就可能过时的注释。

注意**:** 频繁的注释有时反映出代码的低质量。当你觉得被迫要加注释的时候，考虑一下重写代码使其更清晰。

注释不应写在用星号或其他字符画出来的大框里。

注释不应包括诸如制表符和回退符之类的特殊字符。

5.1 实现注释的格式

实现注释的格式主要有4种: block（块）, single-line（单行）, trailing（尾端）, 和 end-of-line（行末）.

5.1.1 块注释

块注释通常用于提供对文件，方法，数据结构和算法的描述。块注释被置于每个文件的开始处以及每个方法之前。它们也可以被用于其他地方，比如方法内部。在功能和方法内部的块注释应该和它们所描述的代码具有一样的缩进格式。

块注释之首应该有一个空行，用于把块注释和代码分割开来，比如：

```
/*
 * Here is a block comment.
 */
```

块注释可以以/*-开头，这样indent(1)就可以将之识别为一个代码块的开始，而不会重排它。

```
/*-
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 *    one
 *        two
 *            three
 */
```

Note: 如果你不使用indent(1)，就不必在代码中使用/*-，或为他人可能对你的代码运行indent(1)作让步。详见于 5.2 节 "Documentation Comments"

5.1.2 单行注释

短注释可以显示在一行内，并与其后的代码具有一样的缩进层级。如果一个注释不能在一行内写完，就该采用块注释(参见"5.1.1 块注释")。单行注释之前应该有一个空行。以下是一个 Java 代码中单行注释的例子：:

```
if (condition) {

    /* Handle the condition. */
    ...
}
```

5.1.3 尾端注释

极短的注释可以与它们所要描述的代码位于同一行，但是应该有足够的空白来分开代码和注释。若有多个短注释出现于大段代码中，它们应该具有相同的缩进。

以下是一个Java代码中尾端注释的例子：

```
if (a == 2) {
    return TRUE;            /* special case */
} else {
    return isPrime(a);      /* works only for odd a */
}
```

5.1.4 行末注释

注释界定符 `//` ，可以注释掉整行或者一行中的一部分。它一般不用于连续多行的注释文本；然而，它可以用来注释掉连续多行的代码段。以下是所有三种风格的例子：

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else
    return false;          // Explain why here.

//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else
//    return false;
```

5.2 文档注释

注意：此处描述的注释格式之范例，参见"Java 源文件范例"。

更多细节，详见"How to Write Doc Comments for Javadoc"，里面包含了文档注释标签的信息(@return, @param, @see):

链接

更多关于文档注释和 javadoc，详见 javadoc 主页

这里

文档注释描述 Java 的类、接口、构造器，方法，以及字段(field)。每个文档注释都会被置于注释定界符 `/**...*/` 之中，一个注释对应一个类、接口或成员。该注释应位于声明之前：

```
/**
 * The Example class provides ...
 */
public class Example {
    ...
}
```

注意顶层 (top-level) 的类和接口是不缩进的，而其成员是缩进的。描述类和接口的文档注释的第一行(`/**`)不需缩进；随后的文档注释每行都缩进1格(使星号纵向对齐)。成员，包括构造函数在内，其文档注释的第一行缩进4格，随后每行都缩进5格。

若你想给出有关类、接口、变量或方法的信息，而这些信息又不适合写在文档中，则可使用实现块注释(见5.1.1)或紧跟在声明后面的单行注释(见5.1.2)。例如，有关一个类实现的细节，应放入紧跟在类声明后面的实现块注释中，而不是放在文档注释中。

文档注释不能放在一个方法或构造器的定义块中，因为 Java 会将位于文档注释之后的第一个声明与其相关联。

PREVIOUS | CONTENTS | NEXT

6 - 声明

6.1 每行声明变量的数量

推荐一行一个声明，因为这样以利于写注释。即,

```
int level; // indentation level
int size;  // size of table
```

要优于

```
int level, size;
```

不要将不同类型变量的声明放在同一行，例如:

```
int foo,  fooarray[]; // 错误!
```

注意: 上面的例子中，在类型和标识符之间放了一个空格，另一种被允许的替代方式是使用制表符:

```
int     level;          // indentation level
int     size;           // size of table
Object  currentEntry;   // currently selected table entry
```

6.2 初始化

尽量在声明局部变量的同时初始化。唯一不这么做的理由是变量的初始值依赖于某些先前发生的计算。

6.3 布局

只在代码块的开始处声明变量。（一个块是指任何被包含在大括号`{`和`}`中间的代码。）不要在首次用到该变量时才声明之。这会把注意力不集中的程序员搞糊涂，同时会妨碍代码在该作用域内的可移植性。

```
void myMethod() {
    int int1 = 0;         // beginning of method block

    if (condition) {
        int int2 = 0;     // beginning of "if" block
        ...
    }
}
```

该规则的一个例外是for循环的索引变量:

```
for (int i = 0; i < maxLoops; i++) { ... }
```

避免声明的局部变量覆盖上一级声明的变量。例如，不要在内部代码块中声明相同的变量名:

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;     // 避免!
        ...
    }
    ...
}
```

6.4 类和接口的声明

当编写类和接口是，应该遵守以下格式规则：

- 在方法名与其参数列表之前的左括号`(`间不要有空格
- 左大括号`{`位于声明语句同行的末尾
- 右大括号`}`另起一行，与相应的声明语句对齐，除非是一个空语句，`}`应紧跟在`{`之后

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}

    ...
}
```

- 方法与方法之间以空行分隔

PREVIOUS | CONTENTS | NEXT

7 - 语句

7.1 简单语句

每行至多包含一条语句，例如:

```
argv++;         // 正确
argc--;         // 正确
argv++; argc--; // 避免!
```

7.2 复合语句

复合语句是包含在大括号中的语句序列，形如 `{ statements }` 。例如下面各段。

- 被括其中的语句应该较之复合语句缩进一个层次
- 左大括号"{"应位于复合语句起始行的行尾；右大括号"}"应另起一行并与复合语句首行对齐。
- 大括号可以被用于所有语句，包括单个语句，只要这些语句是诸如 `if-else` 或 `for` 控制结构的一部分。这样便于添加语句而无需担心由于忘了加括号而引入bug。

7.3 return 语句

一个带返回值的 `return` 语句不使用小括号"()"，除非它们以某种方式使返回值更为显见。例如:

```
return;

return myDisk.size();

return (size ? size : defaultSize);
```

7.4 if, if-else, if else-if else 语句

`if-else` 语句应该是以下形式:

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

注意: `if` 语句通常使用 `{}` 。避免下面容易出错的形式:

```
if (condition) // 避免！这省略了括号{ }！
    statement;
```

7.5 for 语句

`for` 语句应该是如下形式:

```
for (initialization; condition; update) {
    statements;
}
```

空的 `for` 语句 (所有工作都在初始化，条件判断，更新子句中完成) 应该是如下形式:

```
for (initialization; condition; update);
```

当在for语句的初始化或更新子句中使用逗号时，避免因使用三个以上变量，而导致复杂度提高。若需要，可以在 `for` 循环之前(为初始化子句)或 `for` 循环末尾(为更新子句)使用单独的语句。

7.6 while 语句

`while` 语句应该是如下形式:

```
while (condition) {
    statements;
}
```

空的 `while` 语句应该是如下形式:

```
while (condition);
```

7.7 do-while 语句

`do-while` 语句应该是如下形式:

```
do {
    statements;
} while (condition);
```

7.8 switch 语句

`switch` 语句应该是如下形式:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;
case XYZ:
    statements;
    break;
default:
    statements;
    break;
}
```

每当一个 case 顺着往下执行时(因为没有 `break` 语句)，通常应在 `break` 语句的位置添加注释。上面的示例代码中就包含注释 `/* falls through */` 。

Every `switch` statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if later another `case` is added.

7.9 try-catch 语句

`try-catch` 语句应该是如下格式:

```
try {
    statements;
} catch (ExceptionClass e) {
     statements;
}
```

一个 `try-catch` 语句后面也可能跟着一个 `finally` 语句，不论 `try` 代码块是否顺利执行完，它都会被执行。

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

PREVIOUS | CONTENTS | NEXT

8 - 空白

8.1 空行

空行将逻辑相关的代码段分隔开，以提高可读性。

下列情况应该总是使用两个空行：

- 一个源文件的两个片段之间
- 类声明和接口声明之间

下列情况应该总是使用一个空行：

- 两个方法之间
- 方法内的局部变量和方法的第一条语句之间
- 块注释（参见"5.1.1 节"）或单行注释（参见"5.1.2 节"）之前
- 一个方法内的两个逻辑段之间，用以提高可读性

8.2 空格

下列情况应该使用空格：

- 一个紧跟着括号的关键字应该被空格分开，例如：

```
while (true) {
    ...
}
```

注意：空格不应该置于方法名与其左括号之间。这将有助于区分关键字和方法调用。

- 空格应该位于参数列表中逗号的后面
- 所有的二元运算符，除了`.`，应该使用空格将之与操作数分开。一元操作符和操作数之间不因该加空格，比如：负号(`-`)、自增(`++`)和自减(`--`)。例如:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

- `for` 语句中的表达式应该被空格分开，例如:

```
for (expr1; expr2; expr3)
```

- 强制转型后应该跟一个空格，例如:

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

PREVIOUS | CONTENTS | NEXT

9 - 命名规范

命名规范使程序更易读，从而更易于理解。它们也可以提供一些有关标识符功能的信息，以助于理解代码，例如，不论它是一个常量，包，还是类

标识符类型	命名规则	示例
包 (Packages)	一个唯一包名的前缀总是全部小写的ASCII字母并且是一个顶级域名，通常是com，edu，gov，mil，net，org，或1981年ISO 3166标准所指定的标识国家的英文双字符代码。包名的后续部分根据不同机构各自内部的命名规范而不尽相同。这类命名规范可能以特定目录名的组成来区分部门(department)，项目(project)，机器(machine)，或注册名(login names)。	`com.sun.eng` `com.apple.quicktime.v2` `edu.cmu.cs.bovik.cheese`
类 (Classes)	命名规则：类名是个一名词，采用大小写混合的方式，每个单词的首字母大写。尽量使你的类名简洁而富于描述。使用完整单词，避免缩写词(除非该缩写词被更广泛使用，像URL，HTML)	`class Raster` `class ImageSprite`
接口 (Interfaces)	命名规则：大小写规则与类名相似	`interface RasterDelegate` `interface Storing`
方法 (Methods)	方法名是一个动词，采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。	`run()` `runFast()` `getBackground()`
变量 (Variables)	除了变量名外，所有实例，包括类，类常量，均采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。变量名不应以下划线或美元符号开头，尽管这在语法上是允许的。变量名应简短且富于描述。变量名的选用应该易于记忆，即，能够指出其用途。尽量避免单个字符的变量名，除非是一次性的临时变量。临时变量通常被取名为 `i`，`j`，`k`，`m`和 `n`，它们一般用于整型；`c`，`d`，`e`，它们一般用于字符型。	`char c` `int i` `float myWidth`
实例变量 (Instance Variables)	大小写规则和变量名相似，除了前面需要一个下划线	`int _employeeId` `String _name` `Customer _customer`
常量 (Constants)	类常量和ANSI常量的声明，应该全部大写，单词间用下划线隔开。(尽量避免ANSI常量，容易引起错误)	`static final int MIN_WIDTH = 4` `static final int MAX_WIDTH = 999` `static final int GET_THE_CPU = 1`

PREVIOUS | CONTENTS | NEXT

10 - 编程惯例

10.1 提供对实例以及类变量的访问

如果没有充分理由，不要设置实例或者类变量为 public。通常实例变量无需显式的set(设置)和gotten(获取)，通常这作为方法调用的side effect (边缘效应)而产生。

一个具有 public 实例变量的恰当例子是，类仅作为数据结构，没有行为。即，若你要使用一个 `struct` (结构)而非一个类(如果java支持 `struct` 的话)，那么把类的实例变量声明为 public 是合适的。

10.2 引用类变量和类方法

避免用一个对象访问一个类的 (`static`) 变量和方法。应该用类名替代。例如

```
classMethod();             // OK
AClass.classMethod();      // OK
anObject.classMethod();    // 避免!
```

10.3 常量

位于 for 循环中作为计数器值的数字常量，除了 `-1` , `0` 和 `1` 之外，不应被直接写入代码。

10.4 变量赋值

避免在一个语句中给多个变量赋相同的值。它很难读懂。例如:

```
fooBar.fChar = barFoo.lchar = 'c'; // 避免!
```

不要将赋值运算符用在容易与相等关系运算符混淆的地方。例如:

```
if (c++ = d++) {        // 避免! (Java 不允许)
    ...
}
```

应写成

```
if ((c++ = d++) != 0) {
    ...
}
```

不要使用内嵌(embedded)赋值运算符试图提高运行时的效率，这是编译器的工作。例如:

```
d = (a = b + c) + r;        // 避免!
```

应该写成

```
a = b + c;
d = a + r;
```

10.5 其它惯例

10.5.1 圆括号

一般而言，在含有多种运算符的表达式中使用圆括号来避免运算符优先级问题，是个好方法。即使运算符的优先级对你而言可能很清楚，但对其他人未必如此。你不能假设别的程序员和你一样清楚运算符的优先级。

```
if (a == b && c == d)     // 避免!
if ((a == b) && (c == d)) // 正确
```

10.5.2 返回值

设法让你的程序结构符合目的。例如:

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

应该用下面代替

```
return booleanExpression;
```

同理,

```
if (condition) {
    return x;
}
return y;
```

应该写为

```
return (condition ? x : y);
```

10.5.3 条件运算符 `?` 前的表达式

如果一个包含二元运算符的表达式出现在三元运算符 `?:` 的 `?` 之前，那么应该给表达式添上一对圆括号。例如：

```
(x >= 0) ? x : -x;
```

10.5.4 特殊注解

使用 `XXX` 标识处代码虽然实现了功能，但是实现的方法有待商榷，希望将来能改进。使用 `FIXME` 标识处代码需要修正，甚至代码是错误的，不能工作，需要修复。

PREVIOUS | CONTENTS | NEXT

11 - 代码示例

11.1 Java 源文件示例

下面的例子，展示了如何合理布局一个包含单一公共类的Java源程序。接口的布局与其相似。更多信息参见"类和接口声明" and "文档注释"

```
/*
 * @(#)Blah.java        1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */


package java.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Class description goes here.
 *
 * @version 1.82 18 Mar 1999
 * @author Firstname Lastname
 */
public class Blah extends SomeClass {
    /* A class implementation comment can go here. */

    /** classVar1 documentation comment */
    public static int classVar1;

    /**
     * classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;

    /** instanceVar1 documentation comment */
    public Object instanceVar1;
```

```
        /** instanceVar2 documentation comment */
        protected int instanceVar2;

        /** instanceVar3 documentation comment */
        private Object[] instanceVar3;

        /**
         * ...constructor Blah documentation comment...
         */
        public Blah() {
            // ...implementation goes here...
        }

        /**
         * ...method doSomething documentation comment...
         */
        public void doSomething() {
            // ...implementation goes here...
        }

        /**
         * ...method doSomethingElse documentation comment...
         * @param someParam description
         */
        public void doSomethingElse(Object someParam) {
            // ...implementation goes here...
        }
    }
```

PREVIOUS | CONTENTS